From XML to PostScript via XSLT

A lightweight approach for printing course certificates

Michael Piotrowski ⟨*mxp@iws.cs.uni-magdeburg.de*⟩

Otto-von-Guericke-Universität Magdeburg · FIN/IWS

2005-08-04

Abstract This report describes a solution for printing university course certificates (*scheine*). The solution may be considered somewhat unusual, as it directly generates PostScript from XML via XSLT.

1 Introduction

At German universities, students are issued course certificates (known as *scheine*) as proof of attendance. These certificates are typically A5-sized paper slips stating the student name, the title of the course, the credits earned, etc. They are signed by a professor and either handed out to the student or directly sent to the office of examinations. Although they look unassuming, they are very important documents for students, as they are required when registering for exams. It is in the interest of both instructors and students that the certificates are created and issued as quickly and easily as possible: For instructors, to reduce their workload; for students, to ensure that they can meet the registration deadlines.

The author's institution provides templates for course certificates as Microsoft Word documents. Microsoft Word is, however, unsuitable for this task:

1. Printing certificates for a selection of students from a larger number of students based on certain criteria is not a word processing task but really a database reporting task. [1]
2. Some instructors use operating systems for which Microsoft Word is not available.

Unfortunately, the author's institution does not have a central database with student and course information, from which the certificates could be produced automatically. At the beginning of each course, students have to write down their data on paper, and instructors have to key it in. Course information cannot be accessed programmatically either; all necessary information has to be looked up and entered manually, which is both tedious and error-prone.

What is needed as a first step, is a quick, easy and portable way to generate certificates from a list of students. As there is no database available, it should be possible to create this list in a text editor; but it should also be possible to generate it from a database or a spreadsheet, if such a data source is available. Two certificates should be printed on one A4 sheet. The whole process should preferably involve only the invocation of a single command, and it should only

[1] It might be possible to somehow solve it using Word's mail merge facility, but it seems that its functionality would not be sufficient.

depend on a minimum of other software to ensure easy installation, portability and longevity.

2 Approach

For his own use, the author has implemented a solution for printing course certificates based on three industry standards: XML [11] as input format, PostScript [2] as output format, and XSLT [13] for the transformation of the input into the output format. The solution consists of a single XSLT program (typically referred to as *stylesheet*) that transforms an XML input document into a PostScript program (or *page descriptions*) that can be sent directly to a PostScript printer.

This solution satisfies the requirements outlined above: The XML input document can be easily prepared in a text editor or generated from a database. The direct generation of PostScript means that no intermediate formatting step and no separate formatting software is needed. It also means that we have full control over the appearance of the certificates. Besides a text editor (even ***ed*** would be sufficient) and a PostScript printer or a host-based PostScript interpreter (such as Ghostscript or Adobe Acrobat Distiller), the only software that is needed is an XSLT processor. What is more, not only is this process independent of proprietary formats and platforms, but there are also excellent free and open-source implementations of all technologies involved, making it portable, transparent and cost-effective.

3 How does it work?

The stylesheet essentially consists of two parts:

1. The definition of a PostScript procedure (called `schein`) which, given the necessary parameters, renders a single certificate.
2. XSLT code which iterates over the student records of the input file and generates PostScript code which calls the procedure, placing two certificates on one A4 page.

As input, it expects an XML document like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE seminar SYSTEM "seminar.dtd">

<seminar type="hauptseminar">
 <seminarinfo>
  <title>Dokumentverarbeitung</title>
  <credits>6</credits>
  <semester>Wintersemester 2003/2004</semester>
  <certdate>2004-03-03</certdate>
  <field subj="IF">Angewandte Informatik</field>
  <field subj="WIF">Informatik</field>
  <field subj="DKE">Angewandte Informatik</field>
  <field subj="IngIF">Ingenieurinformatik</field>
  <field subj="CV">Praktische und Angewandte Informatik</field>
 </seminarinfo>
```

```
 <subjinfo>
  <subj id="IF">Informatik</subj>
  <subj id="WIF">Wirtschaftsinformatik</subj>
  <subj id="IngIF">Ingenieurinformatik</subj>
  <subj id="DKE">Master in Data and Knowledge Engineering</subj>
  <subj id="CV">Computervisualistik</subj>
 </subjinfo>

 <participants>
  <student sex="f" id="123456" subj="CV" pass="true">
    <fn>Gabriele</fn><sn>Mustermann</sn>
  </student>
  <student sex="m" id="987654" subj="IF" pass="true">
    <fn>Hans</fn><sn>Meier</sn>
  </student>
  <student sex="m" id="876543" subj="DKE" pass="true" grade="1,0 (sehr gut)">
    <fn>Edmund</fn><sn>Schwabacher</sn>
  </student>
 </participants>
</seminar>
```

The XML document consists of two header elements, `seminarinfo` and `subjinfo`. The `seminarinfo` elements contains the information applying to the course as a whole: The course title, the course credits, the semester, and the date of issue. All courses in computer science are assigned to a specific field (theoretical computer science, applied computer science, etc.); the assignment of the course depends on the program in which the student is enrolled (computer science, computational visualistics, business informatics, etc.). The `field` elements provide this information. The `subjinfo` element contains the mapping of program abbreviations to full names.

The two header elements are followed by the list of participating students, contained in the `participants` element; the `student` elements contain all necessary personal information for issuing a certificate. The `pass` attribute indicates whether the student passed the course or not; certificates will only issued to students who passed.

The DTD was kept intentionally simple, but it could easily be extended if necessary. Even though some information could be included from external sources, such as the mappings contained in the `subjinfo` element, it was also an intentional decision to include all necessary information into the file to make it self-contained. This ensures that all information that is current at the date of issue is preserved, and identical certificates can be regenerated later.

Given such an XML file, printing certificates for all the students who passed is then as simple as saying:

```
$ xsltproc scheine.xsl teilnehmer.xml | lp
```

The author uses the ***xsltproc*** XSLT processor from the libxslt package [10], but any other conforming XSLT processor can be used as well. A rendering of the first page of the output generated from the example document above can be seen in Figure 1.

The PostScript output can also be saved, previewed, converted to PDF (“distilled”), etc. When converting to PDF, a bookmark is generated for each

Otto-von-Guericke-Universität
Magdeburg
Fakultät für Informatik

Schein

Herr *Hans Meier* Matrikelnummer 987654
Studiengang Wirtschaftsinformatik
hat im Wintersemester 2003/2004 an der Lehrveranstaltung

Dokumentverarbeitung

erfolgreich teilgenommen.
Zuordung: Informatik Leistungspunkte: 6

Magdeburg, den 2004-03-03 _____
Unterschrift des Verantwortlichen für das Lehrgebiet

Kenntnisnahme durch das **Prüfungsamt**

Magdeburg, den _____
Datum und Unterschrift

Otto-von-Guericke-Universität
Magdeburg
Fakultät für Informatik

Schein

Frau *Gabriele Mustermann* Matrikelnummer 123456
Studiengang Informatik
hat im Wintersemester 2003/2004 an der Lehrveranstaltung

Dokumentverarbeitung

erfolgreich teilgenommen.
Zuordung: Angewandte Informatik Leistungspunkte: 6

Magdeburg, den 2004-03-03 _____
Unterschrift des Verantwortlichen für das Lehrgebiet

Kenntnisnahme durch das **Prüfungsamt**

Magdeburg, den _____
Datum und Unterschrift

Figure 1 | Output of the stylesheet

student; Figure 2 shows the resulting PDF file viewed in Xpdf. The PostScript output is fully compliant with the Document Structuring Conventions (DSC) [1] so that various transformations, such as imposition, reordering, scaling, etc. can be applied to it by a spooler or by utilities like PSUtils [3]. On printers equipped with duplexing capabilities, it is made sure that the output is printed single-sided.

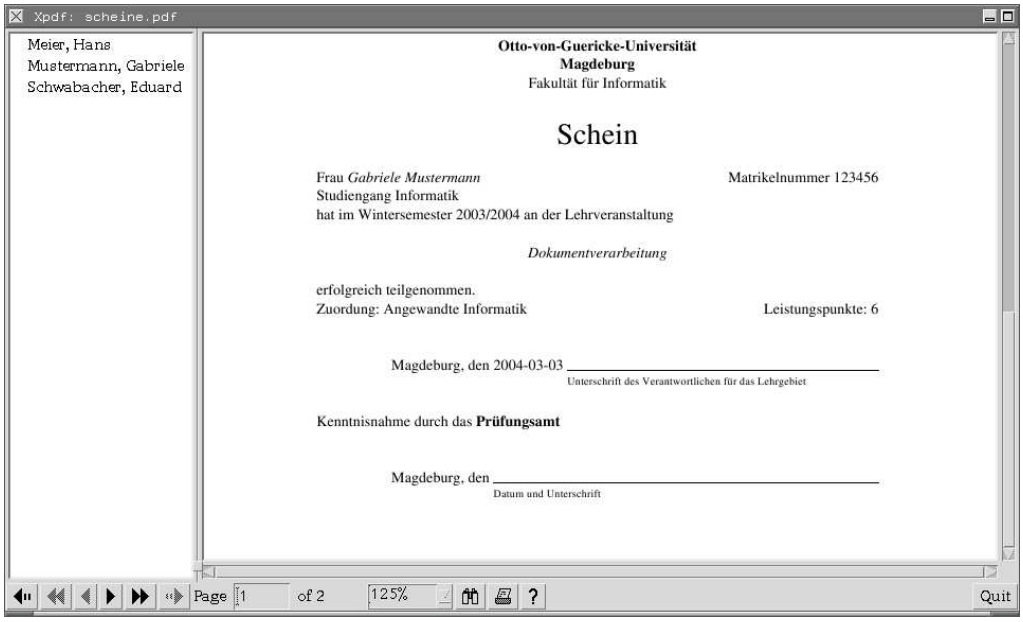

Figure 2 | When converting the PostScript to PDF, bookmarks are created for each student

4 | Implementation Notes

4.1 Sorting and Imposition

Regardless of the ordering of the `student` elements in the input file, the certificates should be sorted alphabetically by the students' last names. However, even simple sorting is a challenge in XSLT 1.0.

The XSLT language is free of side effects. While this has some theoretical benefits (see [5], for example), it is sometimes not very practical. The author has implemented two variants of the first version of the stylesheet: one using recursion, the other using iteration. In the recursive variant it is easy to count the output pages: the template has a `page` parameter, which is incremented with each recursion, so that the DSC `%%Page:` comment can simply be produced like this:

```
<xsl:text>%%Page: </xsl:text>
<xsl:value-of select="$page"/>
```

On the other hand, there is no easy way to sort the `student` elements in this variant. It is easy in an iterative version, though, using `xsl:for-each` and `xsl:sort`, but the generation of the `%%Page:` comment requires cryptic calculations to find out the current page number from the number of the student element that is currently being processed, because one cannot simply use a variable as a page counter:

```
<xsl:when test="position() mod 2 = 1">
  <xsl:text>%%Page: </xsl:text>
  <xsl:value-of select="floor(position() div 2) + 1"/>
```

On closer inspection, however, it turns out that simple alphabetical sorting is not enough. Since two certificates are imposed on one A4 sheet, the imposition should place the certificates in such a way that when the stack of A4 printouts is cut in half, one gets two ordered stacks which can simply be put together to get one stack in alphabetical order; Figure 3 illustrates the desired imposition.

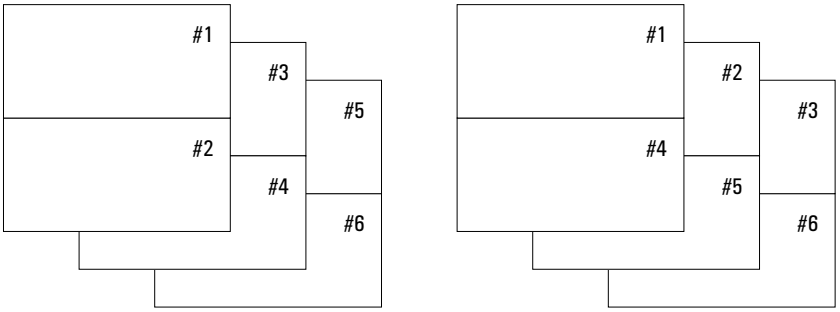

Figure 3 | Left: Straightforward imposition. Right: Desired imposition.

To the author's knowledge this is not possible in pure XSLT 1.0, at least not with reasonable effort. We have therefore implemented a second version of the stylesheet, which uses an EXSLT extension function, *exslt:node-set*(). EXSLT [4], [14], [9] is a community initiative to provide extensions to XSLT. These extensions are supported by most common XSLT processors.

This version of the stylesheet uses two passes: In the first pass, the input elements are sorted alphabetically and the result is assigned to a variable. Even though the content of the variable, a so-called *result tree fragment*, is effectively an XML document, no templates can be applied to it in XSLT 1.0. To be able to treat the variable as input for the second pass, it is necessary to convert it from a result tree fragment to a *node-set*, which is what the *exslt:node-set*() function does.

The second pass basically uses the recursive variant of the first implementation, extended with some code to first select the correct two elements for imposition; the relevant code fragment is:

```
<!-- Select two students to put on one page -->
<xsl:variable name="n" select="round(count($list) div 2) + 1"/>
<xsl:for-each select="$list[position() = 1 or position() = $n]">
```

The parameter `list` contains the sorted list of `student` elements. When the template calls itself in the recursion, the `student` elements that have already been imposed are removed from the `list` parameter:

```
<xsl:call-template name="make-pages">
  <xsl:with-param name="page" select="$page + 1"/>
  <xsl:with-param name="list"
    select="$list[position() > 1 and position() != $n]"/>
</xsl:call-template>
```

4.2 Some General Notes on XSLT

The following are some general observations about XSLT, which were made during the project described in this report, but which are not necessarily specific to this project.

We have already commented on some serious limitations in XSLT 1.0 in Chapter 4.1. Luckily, these limitations will be removed in XSLT 2.0, but currently they make even simple tasks very tedious.

Due to the use of XML syntax, the appearance of XSLT code is incredibly bloated, especially if one can directly compare it to a language like PostScript, as in this project. The verbosity makes the code hard to read. The quoting requirements of XSLT do not help to improve legibility either. For example, to output `<<`, one must either use a CDATA section, such as `<![CDATA[<<]]>`, or character entities, i.e., `&lt;&lt;`. The quoting thus results in an expansion factor between 4 and 7, completely obscuring the original intention.

When generating programming language code, the output must be syntactically valid in the target language; this requires, among other things, the correct usage of whitespace (primarily space and newline characters). Controlling the output of whitespace in XSLT is tedious: It is often necessary to enclose the text to be generated in `xsl:text` elements. While this is similar to the use of string delimiters (usually single or double quotation marks) in other languages, in XSLT the length of the delimiters (21 characters) often exceeds the length of the enclosed string, making the stylesheet even harder to read.

For example, in the follwing piece of code, a PostScript *literal name* (similar to a LISP symbol) is generated using the XSLT function `generate-id()`. [1] A literal name consists of a sequence of characters prefixed by a `/`; there must not be any whitespace between the `/` and the name. To ensure this, even if the line is accidentally (or intentionally) broken at this point, it is necessary to use `xsl:text`:

```
<xsl:text>[ /Dest /</xsl:text><xsl:value-of select="generate-id(.)"/>
```

The author's opinion on XSLT can be summarized as follows:

- The XML syntax inherited from SGML was designed for markup of natural-language text, not for programming languages. While the use of a single syntax for documents and stylesheets might seem compelling at first, it is not really practical. The same seems to apply to XML schema languages, where, for example, a "compact, non-XML syntax" [8] was later added to RELAX NG [7] to "maximize readability".
- The greatest value of XSLT does not lie in its features, but in the fact that it provides a standardized and functionally adequate stylesheet language with multiple independent conforming implementations, something which did not exist for SGML. [2]

[1] XML names, as generated by `generate-id()`, are also legal PostScript names.

[2] Neither CALS FOSI nor DSSSL (ISO/IEC 10179) ever achieved the ubiquity XSLT has today.

5 | Discussion

As stated in the beginning, the stylesheet described above only covers one aspect of course management, namely the printing of course certificates. As long as there is no integrated management of students and courses, its potential advantages are necessarily limited to this aspect.

The author has generated XML input for the stylesheet from an ***sc*** spreadsheet [1] used for the calculation of credit points. One could easily write programs to generate the XML files from other spreadsheet file formats. [2] Of course, all the information in the spreadsheet first has to be entered manually, but it may serve as an example of small-scale integration.

Is the approach appropriate? The answer to this question depends on what one considers "appropriate", so maybe a better question might be, what are the alternatives? There are many alternatives for all three aspects, the input format, the output format and the transformation language. Using a line-oriented input format and ***awk*** as transformation language would be an example of a more "traditional" approach, but nowadays few people would probably question the choice of XML as input format and of XSLT as transformation language – the so-called "XML technologies" are definitely *en vogue*. In this light, what may be questioned, however, is the use of PostScript instead of XSL Formatting Objects (XSL-FO) [12].

Of course, PostScript and XSL-FO are not really comparable: XSL-FO is a declarative specification of what a document should look like, while PostScript is a full programming language; XSL-FO has concepts such as "areas", "margins", "columns" or "leaders", while PostScript has no predefined knowledge of these concepts, but can be used to describe arbitrary layouts, to name just two differences.

So, the question is, in what way would the use of XSL-FO actually improve the stylesheet? The generated XSL-FO document would obviously need to specify the same layout parameters, but the specification would use somewhat higher-level concepts, such as areas and margins instead of coordinates, for example. Despite the higher complexity, this may be seen as an advantage.

The downside is that after having generated an XSL-FO document, it still needs to be printed. This means that it must be rendered to a format that can be sent to the printer (most likely PostScript). This conversion requires an XSL-FO formatter, a complex piece of software, and adds an extra step to the process. The need for an XSL-FO formatter would thus conflict with our goal of minimizing external dependencies.

Furthermore, since the documents – and thus the PostScript page descriptions – in this application are very simple, it is questionable that writing XSL-FO instead of PostScript would have simplified the development or would improve the maintainability. The following description of XSL-FO sums it up quite well: "while powerful, it demands quite a bit of machinery be in place before it begins to work" [6].

In this context the author would like to point out that using "XML technologies" does not mean that problems are somehow solved "automagically": Like other programs, neither XSLT stylesheets nor XSL-FO documents write themselves.

---

[1] ***sc*** is a free curses-based spreadsheet program for UNIX and UNIX-like systems. It is available from http://www.ibiblio.org/pub/Linux/apps/financial/spreadsheet/.

[2] For example, for Microsoft Excel files one could use the Perl Spreadsheet::ParseExcel module.

6 | Conclusion

Course certificates are very simple documents and their production should be equally simple. The solution outlined above works nicely for the author: It only does one job, but it does it well, and it does the job without requiring and depending on hundreds of megabytes of software (be it Microsoft Word, TEX or an XSL-FO formatter), which must be installed, configured and maintained.

The stylesheets described in this report are available from the author.

References

[1] Adobe Systems Incorporated. PostScript Language Document Structuring Conventions Specification. Technical Note #5001 (25 September 1992). Version 3.0.

[2] Adobe Systems Incorporated. *PostScript® Language Reference*. Addison-Wesley. 3rd ed., 1999.

[3] Angus Duggan. PSUtils. URL *http://www.tardis.ed.ac.uk/~ajcd/psutils/*.

[4] EXSLT Home Page. URL *http://exslt.org/*.

[5] Michael Kay. What kind of language is XSLT?: An analysis and overview, 2001. URL *http://ibm.com/developerworks/xml/library/x-xslt/*.

[6] Cameron Laird. RTF on the server: Automate document handling with low-cost server processes. URL *http://ibm.com/developerworks/linux/library/l-sc9/*.

[7] OASIS. RELAX NG Specification. URL *http://www.relaxng.org/spec-20011203.html*.

[8] OASIS. RELAX NG Compact Syntax. URL *http://www.relaxng.org/compact-20021121.html*.

[9] Uche Ogbuji. EXSLT by example, 2003. URL *http://ibm.com/developerworks/xml/library/x-exslt.html*.

[10] Daniel Veillard. Libxslt: The XSLT C library for Gnome. URL *http://xmlsoft.org/XSLT/*.

[11] World Wide Web Consortium. Extensible Markup Language (XML). W3C Recommendation (04 February 2004). Version 1.1. URL *http://www.w3.org/TR/xml11*.

[12] World Wide Web Consortium. Extensible Stylesheet Language (XSL). W3C Recommendation (15 October 2001). Version 1.0. URL *http://www.w3.org/TR/xsl/*.

[13] World Wide Web Consortium. XSL Transformations (XSLT). W3C Recommendation (16 November 1999). Version 1.0. URL *http://www.w3.org/TR/xslt*.

[14] Kevin Williams. XML for Data: Extend XSLT's functionality with EXSLT, 2002. URL *http://ibm.com/developerworks/xml/library/x-xdexslt.html*.